Approximate solutions for second order differential equations with neural networks

Ayush Bahuguna

August 2021

Abstract

Recently, advances have been made in adapting neural networks for approximate solutions to differential equations by employing the dynamics of a targeted differential equation to iteratively optimise parameters in a model. A compilation of established and improved methods is compared in efficacy for solving the inhomogeneous heat equation.

1 Introduction

2 Literature Review

3 Methodology

We desire an approximate solution $\hat{u}(x,t)$ over a selected lattice Ω for the boundary-value problem

$$\begin{aligned} L(u(x,t)) &= f(x,t) \quad \forall (x,t) \in \Omega \\ u(x,0) &= u_0(x) \quad \forall (x,0) \in \delta\Omega \\ u(x,t) &= 0 \quad \forall (x,t), \in \delta\Omega, t \neq 0. \end{aligned} \tag{1}$$

This approximate solution can be realised using a neural approximator with a single hidden layer of the form

$$\hat{u}(x,t) = u_0(x) + \beta(x,t) \sum_{i=1}^{m} v_i \sigma(z_i(x,t)) \tag{2}$$

where β is a constraint function chosen to coordinate the behaviour of the model with boundary requirements, m is the number of artificial neurons, σ is a chosen activation function and z is a linear combination of our inputs

$$z_i = w_i x + k_i t + b_i. \tag{3}$$

The weights and biases v_i, w_i, k_i and b_i comprise our parameter set Θ which has to be optimised in order to realise the required solution, and this optimisation is guided by the dynamics of the targeted differential equation. Therefore we now desire the solution to the minimisation of the square of the PDE residual at each point in Ω

$$min_{\Theta} \quad \frac{1}{|\Omega|} \sum_{(x,t)\in\Omega} (L(\hat{u}(x,t,\Theta)) - f(x,t))^2, \tag{4}$$

where $|\Omega|$ is the cardinality of our lattice. We refer to this residual as our loss function E_Ω. We perform this minimisation by iteratively modifying each element in our parameter set with the gradient descent rule

$$\Delta\theta = -\lambda \frac{dE_\Omega}{d\theta}, \quad \forall\theta \in \Theta. \tag{5}$$

3.1 Residual method

Choosing an adequate function β requires some intuition of the expected solution, since naïvely forcing the model's behaviour in $\delta\Omega$ can conversely affect its freedom in Ω. The simplest approach is to set $\beta(x,t) = 1$ in equation 2, expecting the neural network to satisfy the boundary conditions once idealised approximation is attained. Since minimising the PDE residual does not enforce this behaviour, we now need to calculate the additional residual sum

$$E_{\delta\Omega} = \frac{1}{|\delta\Omega|} \sum_{(x,t)\in\delta\Omega} (L(\hat{u}(x,t)))^2,$$

over the domain boundary, transforming our minimisation problem in equation 4 into

$$min_{\Theta} \quad E_\Omega + \tau E_{\delta\Omega},$$

where τ is a hyperparameter acting as a residual weight for asserting priority between the two loss functions.

3.2 Imposing Boundary Constraints

With prior knowledge of the nature of the desired solution, it is possible to choose an adequate constraint function β without overly limiting our model within our domain. The most straightforward approach is to use $\beta(x,t) = \beta_X(x)\beta_T(t)$ where each separable component independently vanishes as their inputs approach the boundary. One strategy is to define these components as polynomials compliant with such boundary conditions. In the experiments detailed in section **??**, we define all points where $t = 0 \vee x \in \{0,1\}$ as belonging to $\delta\Omega$, hence $\beta_X(x) = x(x-1)$ and $\beta_T(t) = t$ are the simplest polynomial constraints which satisfy the requirements. A more sophisticated approach is to assign one or both of the components as higher degree polynomials. For

example, we explore a d-degree polynomial as our choice for temporal boundary constraint

$$\beta_T(t) = \sum_{j=1}^{d} a_i t^i,$$

where the coefficients a_i are trainable additions to Θ, resulting in a combined regression effort of the polynomial and neural network components. Since the polynomial does not have a constant term it naturally complies with our boundary. However, a caveat of this method is that polynomials necessarily grow unbounded for large inputs, hence it requires a reasonably sized Ω otherwise the polynomial will dominate the dynamics of the model at large inputs; alternatively, the neural network will be forced to vanish at large activations in order to prevent the losses from growing boundlessly. Either outcome is possible depending on the PDE at hand, but mostly on how much Ω spans domain dimensions. This method is hence not suitable for an unbounded lattice, but if kept suitably finite this caveat can be kept controlled if input dimensions are normalised.

3.3 Learning differential operators

An alternative to using intuitively chosen constraint functions is to not learn the solution directly but to instead use our neural network model to approximate a chosen differential operator in the context of the problem statement. This operator should be chosen such that its inverse, the targeted operator L from equation 1 as well as any gradients $(L)_\theta \forall \theta \in \Theta$, should be straightforward to calculate over the neural network in order to attain the desired solution as well as the PDE residual for training. For this reason, the chosen activation function σ should also be well-behaved in order for these terms to be calculated comfortably.

As an example, we experiment with defining the neural network as the approximate velocity $\hat{u}_t$ of the solution in section **??** with

$$\frac{d\hat{u}(x,t)}{dt} = \beta(x) \sum_{i=1}^{m} v_i \sigma(z_i(x,t)), \tag{6}$$

and can then derive our solution by inverting the time derivation with

$$\hat{u}(x,t) = u_0(x) + \beta(x) \sum_{i=1}^{m} \frac{v_i}{k_i} \int_{z_i(x,0)}^{z_i(x,t)} \sigma(\zeta) d\zeta. \tag{7}$$

As is evident, this approach makes it unnecessary to assign the component β_T for our constraint function as continuity with the temporal boundary is implicit in equation 7. In this case, we still require the component β_X to keep the velocity, and equivalently the solution, zero at the spatial boundary. One can consider approximating a further complex operator like $\hat{u}_{tx}$ in order to

avoid using both components, however this poses problems of continuity when the spatial boundary is more complex, besides the issue of having a trickier calculation for the inverse. This issue could possibly be overcome by taking advantage of symmetries in the problem statement, however these are rarely evident or generalisable.

3.4 Recurrent neural networks

In an algebraic context, differential equations always represent a system of recurrence relations. It is then natural to probe recurrent neural networks (RNNs) in the realm of PDE solvers. RNNs act on a sequence of inputs, where the output for a particular term of the sequence is dependent on the network output for previous terms. Similar to how sequential circuits take advantage of input history, this allows these networks to achieve a neural 'memory'. These models have been the focus for significant research in the past decades, with versatile and elaborate architectures being developed.

We explore the simplest formulation of such a network to approximate a solution by restating equation 3 as

$$z_i = w_i x + b_i, \quad t = 0$$
$$z_i = w_i x + k_i t + l_i \hat{u}(x, t - \Delta t) + b_i, \quad t \neq 0$$

where Δt is the temporal step size. This gives us a recurrence for our neural approximator in equation 2 as well as the gradients of model parameters in equation 5.